Asymptotic Analysis (Measure of Performance)

※**Complexity**: amount of resource required by an algorithm, measured as a function of the input size
Time Complexity: number of steps (running time) executed by an algorithm
Space Complexity: number of units of space required by an algorithm (e.g number of elements in a list, number of nodes in a tree/graph)
$t_A(x)$: running time of algorithm A with input x
Worst-Case Running Time: T(n)=max{t(x): x is an input of size n}, the case with the longest running time
Best-Case Running Time: T(n)=min{t(x): x is an input of size n}, the case with the shortest running time
Average-Case running time: expectation of running time, $E[t_n]=\sum_t t\cdot \Pr(t_n=t)$ where t_n is random variable

※**Asymptotic Analysis**: a relationship between the size of input and number of basic operations performed in a long term, order/rate of growth of an algorithm's complexity
We only care about the rate of growth so constant factors don't matter, We only care about the large inputs, so only the highest-degree term matters.
(how well does algorithm perform as the input size grows, n→∞)
※**Basic Operation/Step**: any block of code whose runtime does not depend on the input size
Let f and g be two functions: N→R+
※**Big-Oh** f=O(g) asymptotic upper-bound
function f(x) grows slower or at the same rate as g(x), g is an upper bound on the rate of growth of f
There are constant c>0(c∈R+) and n_0>0(n∈N), such that 0≤f(n)≤c·g(n) for all n≥n_0
e.g. $2n^2=O(n^3)$ $2n^2\in O(n^3)$ (in the set of)
set definition O(g(n))={f(n): there are constant c>0 and n_0>0 such that 0≤f(n)≤c·g(n) for all n≥n_0}
※**Omega** f=Ω(g) asymptotic lower-bound
function f(x) grows faster or at the same rate as g(x), g is a lower bound on the rate of growth of f
Ω(g(n))={f(n): there exist constant c>0 and n_0>0 such that 0≤c·g(n)≤f(n) for all n≥n_0}
e.g. √n=Ω(lg n)
※**Theta** f = Θ(g) asymptotic equivalence, tight bound
function f(x) grows at the same rate as g(x), g has the same rate of growth as f
f=Θ(g) is equivalent to f=O(g) and f=Ω(g), Theta is really an AND of Big-Oh and Omega.
Θ(g(n))=O(g(n))∩Ω(g(n))

Macro Convention:
A set in a formula represents an anonymous function in that set.
$f(n)=n^3+O(n^2)$ means there is a function h(n)=O(n^2) such that f(n)=n^3+h(n), h(n) is a function with lower order terms
e.g. $n^2+O(n)=O(n^2)$ means for any f(n)∈O(n) there is an h(n)∈$O(n^2)$ such that n^2+f(n)=h(n)

For f1∈O(g1(n)) and f2∈O(g2(n)), f1(n)+f2(n)∈O(max{g1(n),g2(n)})
For any polynomial p(n) with degree k, p(n)∈$O(n^k)$. More accurately, p(n)∈$\Theta(n^k)$.
For any function f(n) and positive constant c, cf(n)∈Θ(f(n)).
For positive functions f(n) and g(n), the following holds: f(n)∈Θ(g(n))⇔f(n)∈O(g(n)) and f(n)∈Ω(g(n)), f(n)∈O(g(n))⇔g(n)∈Ω(f(n))
To **prove** a relationship of the form f(n)∈Θ/O/Ω(g(n)):
①using the definition: find a value for c (c_1&c_2), find a value for n0.
Example: polynomial of degree 2 an^2+bn+c is $\Theta(n^2)$ with c1=a/4 c2=7a/4 and n0=2·max(|b|/a, √|c|/a)
②Limit Method: set up a limit quotient $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\begin{cases}0, & then\ f(n)\in O(g(n))\\ c>0, & then\ f(n)\in\Theta(g(n))\\ \infty, & then\ f(n)\in\Omega(g(n))\end{cases}$
If f and g both diverge or converge on zero or infinity, then LHopital rule can be applied.
L Hopital Rule: Let f and g, if the limit between the quotient f(n)/g(n) exists, it is equal to the limit of the derivative f the denominator and the numerator. $\lim_{n\to\infty}\frac{f(n)}{g(n)}=\lim_{n\to\infty}\frac{f'(n)}{g'(n)}$

For the composition ○ of addition, subtraction, multiplication and division, if the limits exist (convergent), then $\lim_{n\to\infty} f_1(n) \circ \lim_{n\to\infty} f_2(n) = \lim_{n\to\infty} f_1(n) \circ f_2(n)$

Worst-Case analysis

—count the number of basic operations(anything whose runtime does not depend on the input size)

①Give a pessimistic upper bound on the number of basic operations that could occur for any input of a fixed size n. Obtain the corresponding Big-Oh expression T(n)=O(f).

②Give a family of inputs (one for each input size),and give a lower bound on the number of basic operations that occurs for this particular family of inputs. Obtain the corresponding Omega expression T(n)=Ω(f).

③Then, conclude that worst-case running time is T(n)=Θ(f)

```
def f(x)
    for item in x
        if item >10
            return 1
    return 1000
end
```

WC=O(n)
For all inputs of size n, the for loop takes n iterations and return statement takes 1 step. The number of steps is less than or equal to (≤) n+1, so WC(≤) n+1, WC=O(n).

WC=Ω(n)
Let x be a list with all items≥10, then f(x) takes more than or equal to (≥) n+1 steps (tracing the code)

algorithm A(x)'s worst-case running time is in $O(n^2)$: argue that, for every input x of size n, the running time of A with input x, i.e. t(x) is no larger than cn^2, where $c > 0$ is a constant.
algorithm A(x)'s worst-case running time is in $\Omega(n^2)$: argue that, there exists an input x of size n, the running time of A with input x, i.e. t(x) is no smaller than cn^2, where $c > 0$ is a constant.
algorithm A(x)'s best-case running time is in $O(n^2)$: argue that, there exists an input x of size n, the running time of A with input x, i.e. t(x) is no larger than cn^2, where $c > 0$ is a constant.
algorithm A(x)'s best-case running time is in $\Omega(n^2)$: argue that, for every input x of size n, the running time of A with input x, i.e. t(x) is no smaller than cn^2, where $c > 0$ is a constant.

Average-Case Analysis

①Define possible set of inputs and a probability distribution of the inputs (prob space for inputs, how inputs are generated, following what distribution)

②Define how to measure runtime (what to count):

③Define a random variable over this probability space to represent the running time of the algorithm.

④Compute expected value of random variable E[T], this is the average runtime: $E[T] = \sum_t t \cdot \mathrm{P}[T = t]$

```
def evens_bad(list)
    if every number in list is even     # check
        repeat list.length times:       # loop
            calculate and print the sum of list
        return 1
    else:
        return 0
```

Check always take n accesses, and loop takes n^2 accesses
①all lists of length n, where item chosen uniformly at random from {1,2,3,4,5} independently
②count list accesses
③Let T be the random variable counting list accesses, T is a function of distribution.
where $T = \begin{cases} n^2+n & \text{input contains only even numbers} \\ n & \text{otherwise} \end{cases}$
④

$$E[T] = \sum_t t \cdot \mathrm{P}[T=t] = (n^2+n)P[T=n^2+n] + nP[T=n] = (n^2+n)\left(\frac{2}{5}\right)^n + n\left(1-\left(\frac{2}{5}\right)^n\right) = n^2\left(\frac{2}{5}\right)^n + n = \Theta(n)$$

Indicator/Dummy/Design/Categorical/Binary/Qualitative Variable, Boolean Indicator

A random variable with value 0 or 1, to indicate the absence or presence of some categorical effect
For any indicator variable X, $E[X] = 1 \cdot P[X=1] + 0 \cdot P[X=0] = P[X=1]$
A common technique is to decompose an arbitrary random variable V into a sum of indicator random variables X1,..Xn. Even when the Xi are dependent on each other,
the expectation of V: $E[V] = E\left[\sum_{i=1}^{n} X_i\right] \underset{\text{expectation}}{\overset{\text{linearity of}}{=}} \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} P(X_i = 1)$
We reduce the problem of computing an expectation to simply computing some probability of (presumably) simpler events.

QuickSort

Pick a pivot and Partition the array, and recursively partition the sub-arrays before and after the pivot.
Each element in the array can be chosen as pivot at most once.
(a pivot never goes into a sub-array on which a recursive call is made)

Elements are only compared to pivots.
(That is what partition is all about—comparing with pivot)

Every pair (a, b) in A are compared with each other at most once.
(The only possible one happens when **a or b** is chosen as a **pivot** and the other is compared to it; after being the pivot, the pivot one will be out of the market and never compare with anyone anymore)

The total number of comparisons is no more than the total number of pairs. $T(n) \le \binom{n}{2} = \frac{n(n-1)}{2} = O(n^2)$

Show $T(n) \in \Omega(n^2)$, worst-case runtime is lower-bounded by $cn^2 \rightarrow$ find one input for which the runtime is at least cn^2.
The worst input for QuickSort is an already sorted array.
So, $T(n)=\Theta(n^2)$. Sorting alg with worst-case runtime n^2: Bubble Sort.
But QuickSort is quick in average-case

```
def QuickSort(array)
    if array.length <2:
        return
    else:
        pivot=array[0]
        smaller, bigger = partition(array[1, :], pivot)
        QuickSort(smaller)
        QuickSort(bigger)
    return smaller+[pivot]+bigger   # array concatenation
def partition(array, pivot)
    smaller=[ ]
    bigger=[ ]
    for item in array:
        if item <=pivot:  # dominating line
            smaller.append(item)
        else:
            bigger.append(item)
    return smaller, bigger
```

①Length of list n, set of inputs is all possible permutations of numbers from 1 to n(inclusive), n! permutations are equally likely (uniform distribution over all possible permutations of {1,…n}) — random permutation
②measure runtime by counting the **number of comparisons** between elements in the list in partition function
Note that the dominating line operation influences the operating time
③Let T be the RV counting number of comparisons

$\forall 1 \le i < j \le n$, Indicator Variable $X_{ij} = \begin{cases} 1 & \text{if } i \text{ and } j \text{ are compared} \\ 0 & \text{otherwise} \end{cases}$

④where total number of comparisons $T = \sum_{1 \le i < j \le n} X_{ij} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$

So $E[T] = E\left[\sum_{1 \le i < j \le n} X_{ij}\right] \xrightarrow{\text{linearity of expectation}} \sum_{1 \le i < j \le n} E[X_{ij}] = \sum_{i<j} P[X_{ij}=1]$

$E[X_{ij}] = 0 \cdot P[X_{ij}=0] + 1 \cdot P[X_{ij}=1] = P[X_{ij}=1]$

Claim: i and j are compared if an only if, among all elements in sorted sub-sequence {i, i+1,…, j}, the first element to be picked as a pivot is either i or j.

Proof:

"**if**": if ***i*** is chosen as pivot (the **first one** among sorted sub-sequence {i, i+1,…, j}), then ***j*** will be compared to pivot ***i*** for sure, because nobody could have possibly separated them yet!

"**only if**": suppose the first one picked as pivot as some k that is between i and j, then i and j will be separated into **different partitions** and will never meet each other.

Claim: $\forall 1 \le i < j \le n$, closed form $P[i \text{ and } j \text{ are compared}] = P[X_{ij}=1]P[X_{ij}=1] = \frac{2}{j-i+1}$

There are j-i+1 numbers in the sorted sub-sequence {i, i+1,…, j}, each of them is equally likely to be chosen as first pivot.

array {1,…n} pivot k, (no idea about orders now) then partition into two arrays {1,…,k-1} & {k+1,…n}
If i & j are compared: i or j are chosen as pivot, i and j in the same partition
1…i…j…n i, i+1,…j-1, j 2/(j-i+1) when j=i+1, i and j are neighbours, always compared

$$E[T] = \sum_{i<j} P[X_{ij}=1] = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n} \sum_{k=1}^{n-i} \frac{1}{k+1} = 2\sum_{k=1}^{n-1} \frac{n-k}{k+1} = 2\sum_{k=1}^{n-1} \left(\frac{n+1}{k+1} - 1\right) = 2(n+1)\sum_{k=1}^{n-1} \frac{1}{k+1} - 2(n-1) = 2(n+1)\Theta(\log n) - 2(n-1) = \Theta(n \log n)$$

Priority Queues and Heaps

Definition of an abstract data type, Implementation of that data type using a particular data structure
※**Abstract Data Type**: (interface) a theoretical model of an entity and the set of operations that can be performed on that entity. (description of what data is stored and operations on data)
specify what the possible operations are for this data type, but say nothing about how the data is stored or how the operations are performed

※**Data Structure**: (implementation) a value in a program which can be used to store and operate on data a representation of data in code(program), tied fundamental to code, and support operations defined in the List ADT, use data structure to implement ADTs.

Example:
Stack is ADT, it stores a list of elements, and supports PUSH(S, v), POP(S), IS_EMPTY(S)
Data structures that can used to implement Stack:
Linked List: Push(insert at the head of the list), Pop(remove at head of the list, if not empty), Is_Empty (return head==None)
Array with a counter for size of stack also works.

List ADT & Array DS

List ADT Operations
①LENGTH(L) return the number of items in L
②GET(L,i) return the item stored at index i in L
③STORE(L,i,x) store the item x at index i in L

Queue ADT

a collection of elements
①ENQUEUE(Q, x)
②DEQUEUE(Q)
③PEEKFRONT(Q)

Priority Queue ADT

a collection of elements with priorities (each element x has x.priority)
Priority Queue Operations
①INSERT(PQ, x, priority) add x to the priority queue PQ with the given priority
②FINDMAX(PQ) return the item in PQ with the highest priority (like PeekFront(Q))
③EXTRACTMAX(PQ) remove and return the item from PQ with the highest/maximum priority

Implement a (Max)-Priority Queue:

Insert(Q, x) insert x at the head
IncreasePriority(Q, x, k) change x.priority to k
Max(Q) go through the whole list
ExtractMax(Q) go through the whole list to find x with max priority, delete it and return it

①Use an unsorted linked list:
Insert takes Θ(1), IncreasePriority takes Θ(1), Max(Q) takes Θ(n), ExtractMax takes O(n)+O(1) = Θ(n).
②Use a reversely sorted linked list:
Max takes Θ(1), ExtractMax takes Θ(1), Insert takes Θ(n) in the worst case, IncreasePriority takes Θ(n) in worst case.
③Heap: Insert Θ(logn), Max Θ(1), ExtractMax Θ(log n), IncreasePriority takes Θ(log n).

※**Binary Tree**: a tree in which every node has at most two children (left & right)
※A binary tree is **complete** if and only if: (structural restriction)
①all of its levels are full, except possibly the last level/bottom one
②all of the nodes in the bottom level (leaves) are as far to the left as possible
A complete tree with n nodes has log(n+1) height.

Heap

A tree satisfies the (max-) **heap properties** if and only if for each node in the tree, the value of that node is greater than or equal to the value of all of its descendants. (every subtree of a heap is also a heap)

A (binary max) **heap** is a binary tree that satisfies the (max-) heap property and is (nearly-) complete.

last leaf =bottom- and right-most node

Importance of being a nearly-complete binary tree:
A complete binary tree can be stored compactly using an array using heap order. Each node i has left child with index 2i, right child 2i+1, and parent floor(i/2). (index starts from 1)
The height of a complete binary tree with n nodes is Θ(log n).

INSERT(Heap, x) Bubble-Up, swap with parent
EXTRACTMAX(Heap) Bubble-Down, swap with elder child
Bubble up/down is also called **percolate/sift/tickle/heapify/cascade** up/down.

```
def INSERT(heap, item):      # tree-based
    put item into next spot in heap, according to completeness property
    while (item > item.parent):
        swap item and item.parent
```

```
def INSERT(heap, x, priority):      # array-based
    heap.size = heap.size +1
    heap[heap.size].item = x
    heap[heap.size].priority = priority
    i = heap.size

    BubbleUp(heap, i)
def BubbleUp(heap, i)
    for i>1 and heap[i].priority>heap[i//2].priority:
        heap[i], heap[i//2] = heap[i//2], heap[i]
        i = i//2
```

WC=O(logn)

A Heap is stored in an array.

```
def EXTRACTMAX(heap):         # tree-based
    move the last leaf to the root of the heap
    curr = heap.root
    while curr<curr.left or curr<curr.right
        swap curr with the larger of its two children
  # when there is no children, stop
```

```
def ExtractMax(heap):         # array-based
    temp = heap[1]
    heap[1] = heap[heap.size]   # replace the root with the last leaf
    heap.size = heap.size-1

    BubbleDown(heap,1)
    return temp

def BubbleDown(heap, i):
    while i<heap.size:
        curr_p = heap[i].priority
        left_p = heap[2i].priority
        right_p = heap[2i+1].priority

        if curr_p >= left_p and curr_p >= right_p:
            break
        else if left_p>right_p:
            heap[i], heap[2i] = heap[2i], heap[i]
            i = 2*i
        else:
            heap[i], heap[2i+1] = heap[2i+1], heap[i]
            i = 2*i+1
```

WC=O(logn)

Sort an array in O(nlogn) time,
ExtractMax for n times, the keys extracted will be sorted in non-ascending order.
Each ExtractMax is O(log n), and we do n times, so overall is O(nlogn).

HeapSort

Modify a max-heap-ordered array into a non-descending sorted array “in-place” (without using any extra array space, just swapping things around)

```
def HeapSort(items):        # sort array into non-descending order
    BuildHeap(items)

    sorted_index = items.length
    while sorted_index > 1:
        swap items[sorted_index], items[1]
        sorted_index = sorted_index -1

        items.length = sorted_index
        BubbleDown(items, 1)

def BuildHeap(items):
    i = items.length
    while i>0:
        BubbleDown(items, i)
        i = i-1
```

```
# Build a heap-ordered array

# swap the first and the last item
# decrement size of heap

# BubbleDown the first element in A
```

```
def BuildMaxHeap(A):
    for i ← floor(n/2) down to 1:
        BubbleDown(A, i)
```

starting index: we start from floor(n/2) and go down to 1
worst-case running time is O(n) instead of O(nlogn)

The height of the heap is logn. There are 2^i nodes at level i.
Number of swaps done by node at level i in the worst case is logn -i
So total number of swaps

$$T(n)=\sum_{i=1}^{n}T(i,n)=\sum_{h=1}^{k}h\cdot\#\ of\ nodes\ at\ height\ h=\sum_{h=1}^{k}h\cdot 2^{k-h}=2^k\sum_{h=1}^{k}\frac{h}{2^h}\xrightarrow{n=2^k-1}(n+1)\sum_{h=1}^{k}\frac{h}{2^h}<(n+1)\underbrace{\sum_{h=1}^{\infty}\frac{h}{2^h}}_{=2}=O(n)$$

Note that $\sum_{k=0}^{\infty}kx^k=\frac{x}{(1-x)^2}$

Binomial Heap

Properties of Binomial Tree:
①B_k tree has 2^k nodes
②degree of root is k; if the children of the root are numbered from left to right by k-1, k-2…0, the child i is the root of a subtree B_i.
③height of the tree is k
④There are exactly $C_i^k=\binom{k}{i}=\frac{k!}{(k-i)!i!}$ nodes at depth i

Binomial Heap (min-heap-ordered)

Dictionaries: AVL Trees & Hash Tables

※ **Dictionary**: a collection of key-value pairs (store associative data with unique ket identifier)
a set S where each node x has a field x.key

Dictionary ADT Operations

①SEARCH(D, key) return the value corresponding to a given key in the dictionary
(return x in D such that x.key=key, return NIL if no such x)
②INSERT(D, key, value) insert a new key-value pair into the dictionary
(if already exists a node with same key, replace it with the new key)
③DELETE(D, key) remove the key-value pair with the given key from the dictionary

Implement a Dictionary:

Search(S, k): return x in S such that x.key = k, return NIL if no such x
Insert(S, x): insert node x into S, if already exists node y with same key , replace y with x

Delete(S, x): delete a given node x from S
Note that Delete(S, key) can be implemented by x=Search(S, k) and then Delete(S, x)

①Unsorted (Doubly) Linked List:
Search takes O(n) worst case, Insert takes O(n) need to check if key is already in the list, Delete takes O(1)
②Sorted Array:
Search takes O(log n) binary search,
Insert O(n), insert at front, everything has to shift to back,
Delete O(n), delete at front, everything has to shift to front
③BST: Search/Insert/Delete takes O(n) in worst-case
④Balanced BST: Search/Insert/Delete takes O(log n) in worst-case

Binary Search Tree: a binary tree that satisfies the BST properties (need not be nearly-complete)
BST property: for every node, its key is ≥ every key in its left subtree, and ≤ every key in its right subtree.

```
def InOrderTraversal(x):      # pass a BST to a function by passing its root node
    # print all keys in BST rooted at x in ascending order
    if x≠Nil:
        InOrderTraversal(x.left)
        print x.key
        InOrderTraversal(x.right)
```

We can perform an in-order traversal to obtain a sorted list from a BST.

Runtime O(n)

```
def Search(D, key):
    if D is empty:
        return None
    else if D.root.key == key:
        return D.root.value
    else if D.root.ket > key:
        return Search(D.left, key)
    else:  return Search(D.right key)
```

key not found, the height of tree is zero

found the key, at D.root

search for left or right subtree
Worst-Case Runtime is O(h), where h is height of tree

```
def Insert(D,key,value):
    if D is empty:
        D.key = key
        D.value = value
    else if D.root.key >= key:
        Insert(D.left, key, value)
    else:   Insert(D.right, key, value)
```

```
if D.root.key == key:
        Insert(D.left, key, value)   # this is an arbitrary choice
```

```
def Delete(D, key):
    if D is empty:
        pass
    else if D.root.key > key:
        return Delete(D.left, key)
    else if D.root.key < key:
        return Delete(D.right, key)
    else:      # D.root.key == key
        if D.left is not empty:
            D.root = ExtractMax(D.left)
        else: D=D.right
```

Runtime analysis with Recurrences:

BST Operations
Read-only
①TreeSearch(root, k): search BST rooted at root, return node with key k, return NIL if not exist
②TreeMinimum(x), TreeMaximum(x): return node with minimum/maximum key of the tree rooted at x, O(h)
③Successor(x), Predecessor(x): find the node which is the successor/predecessor of x in the sorted list obtained by in-order traversal, node with the smallest/greatest key larger/smaller than x
Modifying ④TreeInsert(root, x): Insert node x into the BST rooted at root return the new root of the modified tree; if exists y such that y.key = x.key, replace y with x

Modifying ⑤TreeDelete(root, x): Delete node x from BST rooted at root while maintaining BST property, return the new root of the modified tree

```
def TreeSearch(root, k):
    if root = NIL or root.key = k:
        return root
    if k < root.key:
        return TreeSearch(root.left, k)
    else:
        return TreeSearch(root.right, k)
```

```
def TreeMinimum(x):
    while x.left ≠ NIL:
        x = x.left
    return x
```

TreeMaximum(x) is exactly the same, except that it goes to the right instead of to the left.

```
def Successor(x):
    if x.right ≠ NIL:
        return TreeMinimum(x.right)
    y = x.parent
    while y ≠ NIL and x = y.right:  # x is right child
        x = y
        y = y.parent
    return y
```

Two Cases:
— x has a right child (successor(x) is the minimum of x right subtree)
— x does not have a right child (keep going up to x.parent while x is a right child, stop when x is a left child, then return x.parent if already gone up to the root and still not finding it, return NIL)

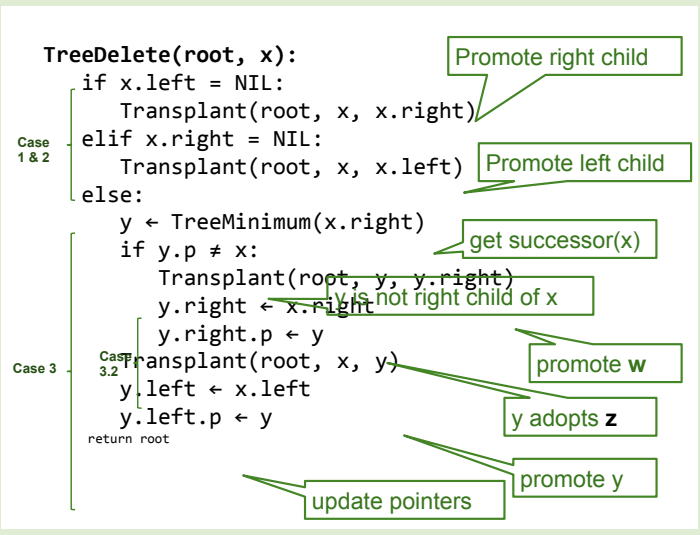
Worst-Case O(h): case 1 TreeMinimum is O(log n)
case 2 at most leaf to root

```
def TreeInsert(root, x):
    if root = NIL:
        root = x
    elif x.key < root.key:
        TreeInsert(root.left, x)
    elif x.key > root.key:
        TreeInsert(root.right, x)
    else: # x.key = root.key
        replace root with x
    return root
```

```
def TreeDelete(root, x):
    if x.left = NIL:
        Transplant(root, x, x.right)       # promote right child
    elif x.right = NIL:
        Transplant(root, x, x.left)        # promote left child
    else: # case 3: get successor(x)
        y = TreeMinimum(x.right)
        if y.parent ≠ x:
            Transplant(root, y, y.right)
            y.right = x.right
            y.right.parent = y
        Transplant(root, x, y)
        y.left = x.left
        y.left.parent = y
    return root

def Transplant(root, u, v):      # v takes away u's parent
    if u.parent = NIL: # u is root
        root = v
    elif u = u.parent.left:
        u.parent.left = v
    elif u = u.parent.right:
        u.parent.right = v
    if v ≠ NIL:
        v.parent = u.parent
```

3 Cases:
— x has no child: just delete
— x has one chid: delete and promote x's only child to the spot together with the only child's subtree
— x has two children: delete the node, promote successor(x)
1.y is x's right child, **promote**
2.y is NOT x's right child:
promote y's right child,
y **adopt** x's right child,
promote y



Height of Tree= length of longest path from root down to leaf in terms of number of edges
Height of a Node=length of longest path from it down to a leaf
The **Balance Factor** of a node in a binary tree is the height of its right subtree minus the height of its left subtree.
A node satisfies the **AVL invariant/property** if its balance factor is between -1 and 1.
A binary tree is **AVL-balanced** if all of its nodes satisfy the AVL invariant/property.
Note: treat AVL tree as augmented BST (augmented: store extra piece of info), every node stores its height

AVL Right Rotation restore the AVL invariant for an unbalanced, left-heavy node

```
def Insert(D, key, value):
        if D is empty
                D.root.key = key
                D.root.value = value
        else if D.root.key == key: #arbitrary choice
                Insert(D.left, key, value)
        else if D.root.ket > key:
                Insert(D.left, key, value)
        else:
                Insert(D.right, key, value)

        Update D.height
        detect and fix Imbalance

def Right_Rotate(D):  # for left heavy node
        # use some temporary variables to match up with the diagram
        y = D.root
        x = D.left.root
        A = D.left.left
        B = D.left.right
        C = D.right

        # Modify references
        D.root = x
        D.left = A
        D.right = AVLTree(y, B, C)
```

Correctness of the AVL Right Rotation

An AVL tree with n nodes has height at most 1.44logn

Hash Tables

Hash table is for implementing Dictionary. Search/Insert/Delete all take O(1).

Direct Addressing directly look up indices (=array)

Universe U: set of all possible keys to store

Hash Function: a function h: U→{0,1,…,m-1} maps U to {0,1,…,m-1}, takes a key and computes the array slot where the key is to be placed in hash table. h(key) is the index in hash table where key is stored

Hash Table: a data structure containing an array of length m (m positions and each position is called a slot/ bucket) and a hash function h: U→{0,1,…,m-1}

```
def Search(hash_table, key):
    hash = h(key)
    return hash_table[hash]
def Insert(hash_table, key, value):
    hash = h(key)
    hash_table[hash] = value
```

```
def Delete(hash_table, key):
    hash = h(key)
    hash_table[hash] = None
```

Collision: if |U|>m, for all L, there exists distinct k_1, $k_2 \in U$, $h(k_1)=h(k_2)$

Perfect Hash Function: if m≥|U|, there are at least as many array slots as possible keys, then there always exists a hash function h which has no collisions.

Closed Addressing/Chaining: store a linked list at each bucket and insert new ones at the head

Each array element stores a pointer to a linked list of key-value pairs. (BST also works, not necessary LL)

```
def Search(hash_table, key):
    hash = h(key)
    linked_list = hash_table[hash]
    if linked_list contains key:
        return corresponding value from linked_list
    else:
        return None
```

```
def Insert(hash_table, key, value):
    hash = h(key)
    linked_list = hash_table[hash]
    insert key, value at the head of linked_list
def Delete(hash_table, key):
    hash = h(key)
    linked_list = hash_table[hash]
    delete key from linked_list
```

Worst-Case: all keys has to the same value/index, all keys hash to the same slot Search, Delete Θ(n)

Average-Case Analysis for closed addressing SEARCH:

①Distribution of inputs:

All possible keys are equally likely to be searched for (uniform distribution on U, assume n random keys are already contained in the hash table)

Simple uniform hashing assumption: hash function h satisfies the property that for all 0≤i＜m, the probability P[h(k)=i]=1/m, when k is chosen uniformly at random from U. (h is fixed hash function, but pick key randomly)

②Measure of runtime T: number of keys compared against the search input + time to compute hash function

③n is number of keys stored in the hash table, m is number of spots in the array

Case 1: key k being searched for is not in the hash table

Alg first computes h(k), then traverse the entire linked list stored at index h(k)

T = 1 + length of linked list at h(k)=1 + number of the n keys that hash to that particular index

By simple uniform hashing assumption(SUHA), the probability that each of the n randomly chosen keys already in the hash table have a hash value of h(k) is 1/m, so n keys hashing to h(k) is n/m

T = Θ(1+n/m)

Case 2: search for k_i, i^{th} key inserted into the hash table (items are inserted at the front of linked list)

n-i kets are inserted after k_i, by the simple uniform hashing assumption, the expected number of keys inserted into $h(k_i)$ after k_i is (n-i)/m

E[T]=1 step computing hash function + average number of keys visited

$$E[T]=1+\frac{1}{n}\sum_{i=1}^{n}\text{expected number of keys visited by search for } k_i = 1+\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{n-i}{m}\right)=2+\frac{1}{n}\sum_{i=1}^{n}\frac{n}{m}-\frac{1}{n}\sum_{i=1}^{n}\frac{i}{m}=2+\frac{n}{m}-\frac{1}{n}\frac{n(n+1)}{2m}=2+\frac{n}{2m}-\frac{1}{2m}=\Theta\left(1+\frac{n}{m}\right)$$

In conclusion, Search and Delete take Θ(n/m+1)=Θ(max(n/m,1))

Load Factor: ratio of the number of keys stored to the size of the table.

α=n/m, n is the number of keys, m is the number of slots

Open Addressing/Probing

each array element to contain only one key, but to allow keys to be mapped to alternate indices when their original spot is already occupied

Parameterized Hash Function h with 2 arguments: h: U×N→{0,…,m-1}

Linear Probing: Given a hash function hash: U →{0,…,m-1} and number b∈{1,…m-1}, parameterized hash function h for linear probing h(k,i)=hash(k)+bi (mod m)
The corresponding linear probing sequence for key k is hash(k,0), hash(k,1), hash(k,2)…
When b=1, the probe sequence is simply hash(k), hash(k)+1…

Problem: Clustering
Quadratic Probing: Given a hash function hash: U →{0,…,m-1} and number b,c∈{1,…m-1}, parameterized hash function h for quadratic probing: $h(k,i)=hash(k)+bi+ci^2$ (mod m)

Double Hashing: Given two hash functions $hash_1$, $hash_2$: U →{0,…,m-1} and number b∈{1,…m-1}, parameterized hash function h for double hashing $h(k,i)=hash_1(k)+b\cdot hash_2(k)$ (mod m)
Note that $hash_1$ and $hash_2$ are two hash functions, independent of each other.

Runtime

Randomized Algorithms

Deterministic Algorithm: number of steps vs input size, probability distribution over the set of possible inputs of a fixed size, then calculate the expected running time over this set of inputs
Randomized Algorithm: In addition to input, algorithm takes a source of random numbers and makes random choices during execution (behaviour can vary even on a fixed input)
The probability (expectation) is only over the random choices made by the algorithm (independent of input)
(no longer assume any special properties of the input) Use randomization to guarantee worst-case expected performance

Note that deterministic algorithm calculates the expected running time across set of inputs with a probability distribution, while randomized algorithm calculate expected running time across the possible random choices made by the algorithm itself with a single input.
Two Types of Randomized Algorithm:
①Las Vegas Alg: deterministic answer, random runtime (randomized-quicksort)
②Monte Carlo Alg: deterministic runtime, random answer (equality testing) gain time efficiency by sacrificing some correctness

For any randomized algorithm, the runtime is a random variable(internal randomness)

Randomized Quicksort

expected worst-case runtime of a randomized algorithm is $ET(n)=\max\{E[T_x] \mid x \text{ is an input of size } n\}$

```
def RandomlyPermute(array):   # Fisher-Yates Shuffle
    for i from 0 to array.length-1
        j = random number between i and array.length -1 (inclusive)
        swap array[i], array[j]
```

For every 0≤i<n, after RandomlyPermute is called, the probability of array[0]=i is equal to 1/n.

```
def RandomlyPermute2(array):
    for i from 0 to array.length-1
        j = random number between 0 and array.length -1 (inclusive)
        swap array[i], array[j]
```

RandomPermute2 cannot generate each permutation with the same probability.

```
def randomized_quicksort(A):
    randomly permute A
    quicksort(A)
```

```
def randomized_quicksort(A):
    quicksort(A)  but each time picking a random element in the array as a pivot
```

We will prove that for any input array of n elements, the expected time is O(nlogn), worst-case expected runtime bound. (no longer assume any special properties of the input)

Let T be RV representing the number of comparisons performed on a sample array drawn from the sample space, now expectation is over the random choices for the pivot and the input is fixed.

Again, $P[i \text{ and } j \text{ are compared}] = P[X_{ij}=1]P[X_{ij}=1] = \frac{2}{j-i+1}$

A different analysis

T(n) is expected time to sort n elements. First pivot chooses i-th smallest element, all equally likely.

Then, $T(n) = (n-1) + \frac{1}{n}\sum_{i=0}^{n-1}(T(i)+T(n-i-1)) = (n-1) + \frac{2}{n}\sum_{i=0}^{n-1}T(i)$ Solve this recurrence gives T(n)≤O(nlogn)

Loop Invariant:

Equality Testing

A holds a binary number x and B holds y, decide whether x=y, let n=len(x)=len(y) be the length of x and y, Randomly choose a prime number p≤n^2, then len(p)≤$\log_2(n^2)=2\log_2(n)$
then compare (x mod p) and (y mod p), return (x mod p) == (y mod p), need to compare at most 2log(n) bits.

Prime number theorem: In range [1, m], there are roughly m/ln(m) prime numbers.
In range [1, n^2], there are n^2/2ln(n) prime numbers. How many bad primes in [1, n^2] satisfy (x mod p)=(y mod p) even if x≠y?
Proof: (x mod p)=(y mod p) ⇔ lx-yl is a multiple of p, p is a divisor of lx-yl

lx-yl <2^n (n-bit binary number) so it has no more than n prime divisors

Out of the n^2/2ln(n) prime numbers we choose from, at most n of them are bad. If choose a good prime, the alg gives correct answer for sure; If choose a bad prime, the alg may give a wrong answer. So the probability of wrong answer P(error) is less than n/(n^2/2ln(n))=2ln(n)/n.

Performance Comparison (n=10TB)
x==y: perform 10^14 comparisons with error probability 0
(x mod p)==(y mod p): perform <100 comparisons with error probability 0.000000000000644
if run Monte Carlo alg twice, perform <200 comparisons with error probability 0.00000000000000000000000215

Universal Hashing

simple uniform hashing assumption(SUHA): for any array index 0≤i<m, the probability of choosing a key which hashes to i is 1/m
Universal Hash Family: Let U be a set of keys and m∈Z^+ be the size of hash table. Let H be a set of hash functions, where each h∈H is a function h: U→{0,1…m-1}. H is a universal hash family if and only if for all pairs of distinct keys k_1,k_2∈U, $P_{h\in H}[h(k_1)=h(k_2)] \le 1/m$
(probability is over picking a random function)

Worst-case expected running time analysis (closed addressing)

Let h∈H be the hash function used by the hash table(chosen uniformly at random)
For each inserted key k_i, let X_i be the indicator random variable which is 1 if $h(k_i)=h(k)$, and 0 otherwise.
Note that the expected values and probabilities are over choices of hash function h, not over choices of keys

$$E\left[\sum_{i=1}^{n}X_i\right] = \sum_{i=1}^{n}E[X_i] = \sum_{i=1}^{n}P[h(k_i)=h(k)] \le \sum_{i=1}^{n}\frac{1}{m} = \frac{n}{m}$$

So the worst-case expected running time for a unsuccessful search with universal hashing is O(1+α), where 1 comes from computing the hash value of k

Construct a universal hash family

Assume U={0,1…,2^w-1} is the set of natural numbers which can be stored on a machine word (w=32 or 64), and m=2^M is a power of two.
For every odd number a, 0<a<2^w and number b(not necessarily odd), 0≤b<2^{w-M}, define hash function:

$$h_{a,b}(k) = \left(ak+b \mod 2^w\right) // 2^{w-M}$$

ak+b: linear shift //2^{w-M}: pick M most significant/left-most bits

Modular Arithmetic

For a positive integer n, two integers a and b are congruent modulo n, $a \equiv b \pmod n$ if the difference $(a-b)$ is an integer multiple of n (n divides $a-b$), where n is the modulus of congruence.
Remainders of the division of both a and b by n are the same.
A is divisible by B = B divides A

Graphs

Vertex/Node v(entity): a single object, circles (labelled or unlabelled, depending on graph context)
Edge (relationship): a tuple $e=(v_1,v_2)$, where $v_1,v_2 \in V$ and $v_1 \neq v_2$ (lines connecting pairs of circles)

A **graph** is a tuple of two sets G=(V,E) where V is a set of vertices and E is a set of edges on those vertices. (A graph is an abstraction of the concept of a set of objects and the relationship between them)

Undirected graph (symmetric): a graph where order does not matter in the edge tuples, $(v_1,v_2)=(v_2,v_1)$ (edge: between two vertices)
Directed graph (non-symmetric relationship): a graph where the tuple order does matter, (v_1,v_2) and (v_2,v_1) represents different edges (edge: from a source vertex to a target vertex)
Measure of graph **size**: number of vertices or edges

Two vertices are **adjacent/neighbours** if there is an edge between them $(v_1,v_2) \in E$.
A **path** between vertices u and w is a sequence of edges (v_0,v_1), $(v_1,v_2)\dots$ (v_{k-1},v_k) where $u=v_0$, $w=v_k$ with all v_i are distinct(no duplicates in sequence).
※ **Length** of path: number of edges in the path. (two adjacent vertices are connected by a path of length 1)
※ **Distance** between two vertices: length of the shortest path between the two vertices. (distance between a vertex and itself is 0)
Graph is **connected** if for every pair of vertices in the graph, there is a path between them.

Graph ADT

VERTICES(G): return a list of all the vertices of G
CHECKAdjacent(G,u,v): return whether vertices u and v are adjacent in G

Adjacent List (vertex focused)

a vertex data type that has a label attribute to identify the vertex and a list of references to its neighbours

```
def CheckAdjacent(G, i, j):
    # Access the vertices attribute and perform lookup by labels
    u = G.vertices[i]
    v = G.vertices[j]
    for vertex in u.neighbours:  # this assumes labels are unique
        if vertex.label == v.label:
            return True
    return False
```

running time: n vertices, m edges
upper bound O(min(m,n)) lower bound Ω(min(m,n))

Space Cost: graph itself stores an array of size Θ(|V|) with each vertex stores its label and references to its neighbours
total cost of storing references is 2=Θ(|E|)
total cost= Θ(|V|+|E|), where |V| is for storing labels

※ **Degree** of a vertex d_v: number of its neighbours

Handshake Lemma: the sum of degrees of all vertices is equal to twice the number of edges $\sum_{v \in V} d_v = 2|E|$

Proof: each edge e=(u,v) is counted twice in the degree sum, once for u and once for v

Adjacent Matrix

n-by-n two-dimensional boolean array, where the-j-entry of the array is true if there is an edge between vertex i and vertex j (n vertices in graph, each with a label between 0 and n-1)

```
def CheckAdjacent(G, i, j):
    return G.adjacency[i][j]
```

Space Cost: Θ(|V|²)
In asymptotic terms, |V|+|E|=O(|V|²) (bound not tight)

Graph Traversal: Breadth-First Search

```
def BFS(graph,s):     # BFS using a queue
    queue = new empty queue
    initialize all vertices in the graph to not enqueued

    queue.enqueue(s)
    s.enqueue = True
    while queue is not empty:
        v = queue.dequeue()
        Visit(v)     # do sth with v, like print out its label
        for each neighbour u of v:
            if not u.enqueued:
                queue.enqueue(u)
                u.enqueued = True
```

Runtime
creation of new queue takes constant time
initialization of all vertices to "not queued" takes $\Theta(|V|)$ time
Outer Loop runs at most $|V|$ times.
Proof: No vertex is added to the queue more than once and at each iteration, only one item is dequeued.
Inner Loop runs at most $2|E|$ times
(analysis same as space cost analysis of adjacency list)
inner loop iterates once per neighbour of current vertex, and current takes on the value of each vertex at most once, so the total number of iterations of the inner loop across all vertices is bounded above by the total number of neighbours for each vertex, which is $2|E|$ by Handshake Lemma.
Upper Bound on the worst-case runtime $O(|V|+|E|)$

Correctness of BFS
G=(V,E), starting vertex v for BFS, let $w \in V$, connected to v ($\exists$ path between v and w):
①BFS visits w
②Let d be the distance between v and w, then w is visited after every vertex at distance at most d-1 from w.
$\forall\ u \in V$ with distance d' from v, d'<d. Then BFS visits u before w.
Proof: let P(n) be the statement that 1 & 2 are true for all vertices with distance n from v
Base Case P(0): the only vertex at distance 0 from v is v itself, 1 is true because v is visited, it is added to the queue at the beginning of algorithm. 2 is vacuously true because there are not even two distinct vertices to consider.
Inductive Step: induction hypothesis let $k \geq 0$ assume P(k) holds
(all vertices at distance k from v are visited and visited after all vertices at distance at most k-1)
—If there are no vertices at distance k+1 from v, P(k+1) is vacuously true.
—Statement 1:
Let w be a vertex at distance k+1 from v, by definition of distance, w must have a neighbour w' at distance k from v (w' is the vertex immediately before w on the path of length k between v and w). By induction hypothesis, w' is visited. When a vertex is visited, all of its neighbours are enqueued into the queue unless they have already been enqueued. So w is enqueued. Because the loop does not terminate until the queue is empty, this ensures that w at some point becomes the current vertex and hence is visited. Statement 1 holds for w
—Statement 2:
Let u be a vertex at distance d from v ($d \leq k$) where d=0 or $d \geq 1$
$d \geq 1$ case: u has a neighbour u' at distance d-1 from v which is visited before u by induction hypothesis. Also, u' is visited before w' since w' is at distance k from v and u' is at distance $d-1 \leq k-1$ from v. Then the neighbours of w' which include u, must be enqueued before the neighbours of w', which include w. So u is added to the queue before w and hence is dequeued and visited before w.

Analysis of BFS
No vertex is added to the queue more than once.
Proof: every time a vertex is added to the queue, it is always marked as 'enqueued'. Only vertices which are not enqueued are ever added to the queue—the starting vertex is initially unnoticed and there is a check inside the inner loop before enqueuing new vertices, so every vertex is added to the queue at most once.

Graph Traversal: Depth-First Search

```
def DFS(graph, s):      # DFS using a stack (one attribute .pushed)
    stack = new empty stack
    initialize all vertices in the graph to not pushed
    stack.push(s)
    s.pushed = True
    while stack is not empty
        v = stack.pop()         # pop: remove sth from stack
        Visit (v)    # do sth with v, like print its label
        for each neighbour u of v:
            if not u.pushed:
                stack.push(u)
                u.pushed = True
```

```
def DFS(graph, s):
  # using recursion, with two attributes (. started & .finished)
    initialize all vertices in the graph to not started or finished
    DFS_helper(graph, s)
def DFS_helper(graph, v):
    v.started = True
    Visit (v)
    for each neighbour u of v:
        if not u.started:
            DFS_helper(graph, u)
    v.finished = True
```

Not Started: the vertex has not been visited by Search
Started but Not Finished: has been visited by Search, but some vertices reachable from this vertex without revisiting other started vertices still need to be visited.
Started and Finished: has been visited by Search, and all vertices reachable from this vertex without revisiting other started vertices have been visited.

Weak **Correctness** of DFS
Let v be the starting vertex given as input the DFS algorithm. Then for every vertex w which is connected to v, w is visited.

Suppose DFS_helper is called on a vertex v, and let w be any vertex in the graph which satisfies the following two properties at the beginning of the function call: w is not visited; there is a path between v and w which consists of only vertices which have not been started.
Then w is both started and finished when DFS_helper(v) returns.

The worst-case runtime of depth-first search is $\Theta(|V|+|E|)$

DFS Forest

Tree Edge: edge traversed when DFS (those edges form a tree)
Back Edge: point from a node to one of its ancestors in the DFS tree
Forward Edge: point from a node to one of its descendants
Cross Edge: point from a node to a previously visited node that is neither an ancestor nor a descendant

Application of Graph Traversal: detect cycles in a graph

```
def BFSwithParent(graph,s):     # augmented form of BFS
    queue = new empty queue
    initialize all vertices in the graph to not enqueued
        and all parent attributes to null

    queue.enqueue(s)
    s.enqueue = True
    s.parent = null

    while queue is not empty:
        v = queue.dequeue()
        Visit(v)     # do sth with v, like print out its label
        for each neighbour u of v:
            if not u.enqueued:
                queue.enqueue(u)
                u.enqueued = True
                u.parent = v
```

Graph with .parent attribute:
v is the first neighbor of u to be visited
v is the vertex causing u to be enqueued/visited

Suppose we run BFSwithParent starting at s, let v be any vertex connected to s. Then the "chain of parent" from v is a shortest path between s and v.

Cycle: a sequence of distinct edges $(v_0,v_1),(v_1,v_2)\ldots(v_{n-1},v_n), (v_n,v_0)$ which start and end at the same vertex.
(cycle encodes a natural redundancy in a graph, it is possible to remove any edge in a cycle without disconnecting its vertices)

Detect cycles in undirected graph: Modified DFS (only works when the graph is connected)
—each vertex stores its parent vertex, which is the neighbour causing the vertex to be visited
—inner loop contains a check for started neighbours that are not the parent of the current vertex

```
def DetectCycle(graph, s):
    initialize all vertices in the graph to not started or finished
        and all parents attributes to null
    s = pick starting vertex
    DFS_helper(graph, s)
def DFS_helper(graph, v):
    v.started = True
    for each neighbour u of v:
        if not u.started:
            u.parent = v
            DFS_helper(graph, u)
        else: # u was already started
            if u is not parent of v:
                u and v are part of a cycle
        # if u is the parent of v, it is just a single edge
    v.finished = True    # important for directed graph
```

Correctness of cycle detection with DFS:
The modified DFS alg reports the existence of a cycle if and only if the input graph has a cycle.
Proof:
①Forward Direction: assume alg reports a cycle, prove that the input graph has a cycle
②Backward Direction: assume the input graph has a cycle, show that a cycle is reported by this alg

Detect cycles in directed graph:
—a neighbour of v is another vertex u such that there is an edge from v to u
—a directed cycle can have length 2: if there are two vertices u and v such that there is an edge from u to v and one from v to u, there two vertices form a cycle

```
def DetectCycleDirected(graph, s):
    initialize all vertices in the graph to not started or finished
        and all parents attributes to null
    DFS_helper(graph, s)
def DFS_helper(graph, v):
    v.started = True
    for each neighbour u of v:
        if not u.started:
            DFS_helper(graph, u)
        else:
            if not u.finished:
                report there is a cycle (directed)
    v.finished = True
```

```
# if u.finished == False, we are still in the recursive call of u
```

Application of Graph Traversal: determine the shortest path between two vertices

A spanning tree of a connected graph G(V,E) is a graph T=(V, E') where E'≤E (E' is a subset of original edges), such that ①T is still connected ②T has no cycles
BFS with parent gives a spanning tree.

A weighted graph is the same as before except edges are triples (u,v,w) where u, v are pair of vertices u,v∈V, and w∈R (weight/cost of edge) E={(u,v,w) | u,v∈V,w∈R}
Minimal Spanning Tree: (remove redundant edges but still keep the graph connected)
The minimal-(weight) spanning tree (MST) problem is input—a weighted graph, output—a spanning tree T=(V, E'), minimizing $\sum_{e\in E} w_e$

|V|=|E|−1, can stop at |E|=|V|−1

```
1.  def PrimMST(G):
2.      v = pick starting vertex from G.vertices
3.      TV = {v}            # MST vertices
4.      TE = {}             # MST
5.      while TV != G.vertices
6.          extension = {(u, v) in G.edges where exactly one of u, v are in TV}
7.          e = edge in extension having minimum weight
8.          TV = TV + endpoints of e
9.          TE = TE + {e}
10.     return (TV, TE) tuple
```

while loop |V|-1 iterations: at each iteration, the size of TV increased by exactly 1

6. if both u & v in TV (it forms a cycle — redundancy)
6. extension loop through all edges
6/7. extension & e takes O(|E|) naively

8.TV size +1 at each iteration
9.TE = TE+{e} only add, never remove, never back track
O(|V|·|E|)

Build up a tree that starts small and eventually covers every vertex,
loop invariant for Prim: at the start of each loop iteration
①(TV, TE) is a tree, connected and no cycles
②(TV, TE) (partial solution) can be extended by adding some vertices and edges to an MST (full solution)

Extension does not change very much:
—use a heap to store edges (course note)
—use a heap to store vertices (CLRS)
—use a Fibonacci heap to store vertices (CLRS)

```
def KruskalMST(G):
    TE = {}
    sort G.edges in non-decreasing order of weights/costs
    set each vertex to be its own connected component

    for e in G.edges:
        if the endpoints of e are not in the same connected component:
            TE = TE + {e}
            merge the connected components containing the endpoints of e
    return (graph.vertices, TE)
```

For Kruskal, invariant ② also applies

Disjoint Set: used to store connected components for Kruskal or other things

Amortized Analysis

bulk operations: a sequence of operations on data structure

※**Worst-Case Sequence Complexity** of a sequence of m operations: the maximum possible total time cost over all possible sequences of m operations (similar to worst-case time for one operation)

We do **Amortized Analysis** when we are interested in the total complexity of a sequence of operations. (unlike in average-case analysis where we are interested in a single operation)

※**Amortized Sequence Complexity**: average cost per operation over the sequence (unlike average-case analysis no probability or expectation involved)

$$Amortized\ Sequence\ Complexity = \frac{worst-case\ Sequence\ Complexity}{number\ of\ operations\ m}$$

Methods for Amortized Analysis: Aggregate Method; Accounting Method; Potential Method

Accounting Method

Find a charge (some number of time units charged per operation) such that: the sum of the charges is an upper bound on the total actual cost.

Like maintaining a bank account, low cost operations charged a bit more than their actual amount; the surplus is deposited in the account for later use.

—Assign a charge for each operation

—When charge > actual cost, leftover amount is assigned as credit (usually to specific elements in data structure)

—When an operation's charge < actual cost, use some stored credit to "pay" for excess cost.

—For this to work, need to argue that credit is never negative

If we have more than one operation, we can assign different charges to each one

Binary Counter

sequence of k bits(k is fixed), single operation Increment adds 1 in binary and the cost of one Increment is number of bits that need to change.

If we do n Increments, worst-case complexity of any Increment is logn.

Naive analysis: worst-case complexity of n increments is nlogn (bad, worst case only happens once)

1.Aggregate Method

Initially	"0000"	Cost	i-th bit changed	……	bit	changes	total number of changes
1.Add 1	"0001"	1	1		1	every operation	n (n=# of operations)
2.Add 1	"0010"	2	1, 2		2	every 2 operations	n/2
3.Add 1	"0011"	1	1		3	every 4 operations	n/4
4.Add 1	"0100"	3	1, 2, 3		4		
5.Add 1	"0101"	1	1		…	…	
6.Add 1	"0110"	2	1, 2		i	every 2^i operations	n/2^(i-1)
7.Add 1	"0111"	1	1				
8.Add 1	"1000"	4	1, 2, 3, 4				

$$\text{Total Number of Bit Flips during Sequence} = \sum_i (flips\ of\ bit\ i) = \sum_{i=1}^{n} \frac{n}{2^{i-1}} = n\left(1+\frac{1}{2}+\frac{1}{4}+...+\frac{1}{2^{\log n}}\right) \le 2n$$

So Amortized Cost ≤2n/n=2 for each operation.

2.Accounting Method

—Charge each operation $2

$1 to flip 0 to 1 (only one bit flips from 0 to 1); use stored credits to pay for flips 1 to 0.

$1 credit stored with the bit just changed to 1

Credit Invariant: at any step, each bit of the counter that is equal to 1 will have $1 credit

Proof: (by induction)

Base: initially counter is 0 and no credit

Induction Step: assume true up to some value of x and now consider next step

case ① x = b..b b 0 1..1 → b..b b 1 0..0 (i least significant bits are 1, i+1st bit is 0)

actual cost=i+1 used i credits to pay for i flips 1 to 0, use 1(out of 2) to pay for 0 to 1, use 1(out of 2) for credit on the new "1" at i+1st.

case ② x = 1 1..1→0 0..0 (all bits are 1) actual cost is k, use k credits to pay for k flips 1 to 0, no need extra $2

Credit Invariant is always true, so total charge for sequence is upper bound on total cost.

(need invariant to show that the credit is always positive)

Total Charge = 2n, so amortized cost per operation=2

Dynamic Array: resize the array (make it bigger)

array are fixed blocks of memory. 2 cases: size < allocated, size = allocated (dynamic array handle this)
attribute: —allocated: number of spots available
—size: number of occupied spots (# of items in the array)

```
def INSERT(A, x):        # insert at the end of the occupied section
    if A.size < A.allocated:
        A[A.size] = x                                          # fill in one more spot
        A.size ++                                              # insert new item: 1 time array access
    else:
        allocate new space for A, with A.allocated ×2 spots
        copy old element over                                  # copy: n reads + n writes
        insert x
```

Runtime: m = A.allocated n = A.size
T(n)= number of array accesses for INSERT (a function of n, count read and write separately)

Array are initialized with 1 spot allocated. (n must be a power of 2)

$$T(n)=\begin{cases}1, \ if \ n<m\\ 2n+1, \ if \ n=m\end{cases}\text{(not count time consuming of allocate new spaces)}$$

copy: 2n (n reads, n writes) insert: 1 array access
T(n) = 2n+1, if $n=2^k$ 1, otherwise
INSERT in the worst case takes O(n)

Aggregate Analysis

Start with empty array (size=0, alloc=1)
Perform M INSERTs (“sequence”)

Total Cost $\sum_{n=0}^{M-1} T(n) \xrightarrow{T'(n)=T(n)-1=\begin{cases}2n \ if \ n=2^k\\ 0 \ otherwise\end{cases}} \sum_{n=0}^{M-1} 1+T'(n) = M+\sum_{n=0}^{M-1} T'(n) = M+\sum_{k=0}^{\lfloor \log(M-1)\rfloor} 2\cdot 2^k \le M+2\cdot 2M = 5M$

M INSERTs takes ≤5M accesses O(M)
The average cost of an/per INSERT is ≤5M/M=5 O(1)
Note: average here is not talking about probabilistic
average—take a branches of them, add them up and divided by the number of time

Now allow insertion & selection in sequence of operations
DELETE if array is too big(too many empty spot), contract the array

Aggregate Cost: the total time for M operations (worst case of sequence operations) in the worst case
(max over all sequences in M operations)

Potential Method

In amortized analysis, we want to find (a good upper bound on) the maximum possible total cost of a sequence of M operations. For a fixed operation sequence, we define the following variables:
$D_0, D_1, \dots D_M$ represent the states of the data structure, with **operation i** transforming state D_{i-1} to state D_i.
D_0: **initial state** of the data structure (e.g empty dynamic array)
D_M: final sate of the data structure after all M operations have been performed

C_i: actual cost of the i^{th} operation Total cost of the sequence: $\sum_{i=1}^{M} C_i$

Amortized Cost of the i^{th} operation: $\hat{C}_i$
Want to find values for the amortized cost such that:
①each $\hat{C}_i$ is “small” (e.g constant with respect to M, O(1))

②total **amortized cost** is an upper bound on the total real cost $\sum_{i=1}^{M} C_i \le \sum_{i=1}^{M} \hat{C}_i$ implied by $\varphi(D_M) \ge \varphi(D_0)$.

Potential Function ϕ: D→R a map from a state to a real number, D: all possible states

Define **Amortized Cost** of each operation: $\hat{C}_i = C_i + \varphi(D_i) - \varphi(D_{i-1})$

(take the original cost C_i and offset it by the change/difference in "potential energy stored" in the data structure)

If C_i is small, allow $\Phi(D_i)$ to be (a bit) greater than $\Phi(D_{i-1})$ and keep the amortized cost small;

If C_i is large, offset it if $\Phi(D_i)$ is smaller than $\Phi(D_{i-1})$ by roughly the same amount.

Requires two properties of potential function $\Phi(D_i)$ for all i:

①$\Phi(D_{i-1}) - \Phi(D_i)$ is roughly $\geq C_i$ (offset high costs)

②$\Phi(D_i) \geq \Phi(D_0)$, which ensures that the total amortized cost will be ≥ the total actual cost

$$\sum_{i=1}^{M} \hat{C}_i = \sum_{i=1}^{M} \left[C_i + \varphi(D_i) - \varphi(D_{i-1}) \right] = \sum_{i=1}^{M} C_i + \varphi(D_M) - \varphi(D_0) \qquad \sum_{i=1}^{M} \hat{C}_i \geq \sum_{i=1}^{M} C_i \ \ if \ \ \varphi(D_M) - \varphi(D_0) \geq 0$$

Potential Method allow to analyze a sequence of operations by studying the local change a single operation causes, regardless of what operations have come before in the sequence.

Example: Dynamic Array INSERT with potential function $\varphi(D) = 2\left|2D.size - D.allocated\right|$

Suppose operation i is an INSERT, in which the array is full (causing an expansion).

(right before the operation)D_{i-1} with size n_{i-1} and amount of allocated space m_{i-1} and $n_{i-1} < m_{i-1}$,

(right after the operation INSERT)D_i with size n_i, allocated space m_i

Relationship between them: $n_{i-1}=m_{i-1}$, $n_i=n_{i-1}+1$, $m_i=2m_{i-1}$.

cost of an INSERT in this case: $C_i=2n_{i-1}+1$ (count 2 array accesses per item in D_{i-1} to copy it and 1 more to store the new item)

$$\hat{C}_i = C_i + \varphi(D_i) - \varphi(D_{i-1})$$
$$= (2n_{i-1}+1) + 2\cdot\left|2\cdot n_i - m_i\right| - 2\cdot\left|2\cdot n_{i-1} - m_{i-1}\right|$$
$$= (2n_{i-1}+1) + 2\cdot\left|2\cdot(n_{i-1}+1) - 2n_{i-1}\right| - 2\cdot\left|2n_{i-1} - n_{i-1}\right|$$
$$= 5$$

Example: Incorporating DELETE

```
# Delete the last item from the array, and resize if necessary
array[array.size] = null
array.size -= 1

# Resize
if array.size <= array.allocated/4:   # ceiling
        create a new array with (array.allocated/2) spots
        copy items over
```

breakpoint: resize if the number of array elements is one-quarter of the total allocated space, and cut the number of allocated spots in half

stable state→one-half full (potential=0)

Assume the number of allocated spots is a multiple of 4, the resize check in the DELETE algorithm ensures that over 1/4 of the allocated spots are always occupied:

$$C_i = \begin{cases} 1, \ if \ \ n_{i-1} > \dfrac{m_{i-1}}{4} + 1 \\ 2(n_{i-1} - 1) + 1 = 2n_i + 1, if \ \ n_{i-1} = \dfrac{m_{i-1}}{4} + 1 \end{cases}$$

Disjoint Set

n different items, partition into disjoint sets (intersection is empty between sets)

Each set has a **set representative** (a unique item in the set used to identify set)

Note: set does not have orders (we don't care about orders), pick one set representative randomly.

We can use single linked list to store disjoint set (have no idea where is the endpoint of linked list, since we can only trace upwards to fine the set representative)

Union merges two linked lists into a tree, where x and y are nodes in LL.

```
def Makeset(DS, v):
        node = new Node with value v
        return node

def Find(DS, x):   # return the set representative of x's set
        curr = x
        while curr.parent is not null:
                curr = curr.parent
        return curr

def Union(DS, x, y):      # merge x's set and y's set, x y are items
        root1 = Find(DS, x)
        root2 = Find(DS,y)
        # root1 and root2 are the roots of their respective trees
        root1.parent = root2
```

if Find(x1)==Find(x2), x1 and x2 are in the same set

The **Rank** of a node in a disjoint set is defined recursively: rank of a leaf is 0; rank of an internal node is one plus the maximum rank of its children. (different from tree height)

```
def MakeSet(DS, v):
        node = new Node with value v
        node.rank = 0       # set the rank
        return node

def UnionByRank(DS, x, y):
        root1 = Find(DS, x)
        root2 = Find(DS, y)

        if root1.rank > root2.rank:
                root2.parent = root1
        else if root1.rank < root2.rank:
                root1.parent = root2
        else:
                root1.parent = root2
                root2.rank = root2.rank + 1
```

root1 ad root2 are the roots of their respective trees

choose the one with the larger rank as the new root

only need to increment a rank when the two root ranks are equal
Ensure that the merged tree's height grows only when two trees of equal height are merged

Find operation takes a long time, because input node is far from the root of the tree and all of the node's ancestors must be traversed.

Path Compression can reduce the height of the tree, but rank of each node remains unchanged (rank only updated by Union operations)

```
def Find(DS, x):            # Find with Path Compression
        root = x
        # After this loop, root is the root of the tree containing x
        while root is not null:
                root = root.parent

        # Path Compression: set all ancestors of x to be children of root
        curr = x
        while curr is not null and curr.parent is not null:
                next_node = curr.parent
                curr.parent = root
                curr = next_node
        return root
```